ERDC MSRC/PET TR/00-32

Practical Experiences with the Fortran Pthreads API

by

Clay P. Breshears
Phu Luong

13 July 2000

07h0132000

Abstract

With the growing popularity of symmetric multiprocessors (SMPs), shared-memory programming models have become more important. Of particular relevance to scientific programmers will be those paradigms that can be used within message-passing codes. POSIX Threads (Pthreads) is one such shared-memory programming model. While only defined for use within C programs, the Fortran API to Pthreads (FPTHRD) developed at the ERDC MSRC gives Fortran programmers access to the Pthreads library functions.

During development of the Fortran API, small problems, focused on very specific functionality, were coded and run to demonstrate the potential effectiveness of Pthreads on numerical computations. Taking the next logical step, we have applied threaded and concurrent programming approaches to a complete, production-level application using the FPTHRD package on the SGI Origin 2000 at the ERDC MSRC. The code used in this study is a multiblock grid version of the Princeton Ocean Model (MGPOM) coastal ocean circulation model. This code uses MPI for processing individual grid blocks on separate processors and sharing of data between blocks.

We describe the process used to modify the MGPOM code for concurrent computations within MPI processes. This includes profiling to identify sections of code that could benefit from threading and dependency analysis of loops included within selected routines. Preliminary tests with only a single subroutine of the MGPOM code modified for threaded computation have shown a 15% speedup. Code modifications to achieve these results took less than 15 minutes. We expect further execution time reductions as other routines are modified for concurrent execution.

Practical Experiences with the Fortran Pthreads API

Clay P. Breshears[1] Phu Luong[2]

July 13, 2000

[1]Rice University; PET On-site Scalable Parallel Programming Tools Lead
[2]University of Texas, Austin; PET On-site Environmental Quality Modeling Lead

1 Introduction

Pthreads is a POSIX standard library [4] for expressing concurrency on single processor computers and symmetric multiprocessors (SMPs). Since many scientific computations contain opportunities for exploiting functional, or task-level concurrency, many Fortran applications would benefit from multithreading. However, as useful as the Pthreads standard is for concurrent programming, a Fortran interface is not defined. This deficiency was rectified with the development of a Fortran 90 API to Pthreads (FPTHRD) [3].

For many programmers that have been working with distributed memory models, such as MPI, shared memory, multithreaded programming may be unfamiliar. While the data decomposition used within a problem may be similar between the two models, the sharing of data and synchronization—handled automatically via message passing routines—between threads is the responsibility of the programmer. Fortunately there are standard programming techniques that can be used to facilitate sharing and synchronization between threads.

We shall examine some of these standard methods and apply them to a production quality ocean circulation model. This code, MGPOM, is a modification of the Princeton Ocean Model (POM) that incorporates a multiblock grid [1]. In Section 2 we present a brief outline of the code and how multiblock grids have been applied to it. Section 3 gives a summary of the relevant functionality within the Pthreads library and details of the FPTHRD package. Section 4 details how to apply Pthreads to existing codes. Examples from the development and implementation of the multithreaded MGPOM code will be used for illustration. The performance improvements from running this threaded code are given in Section 5.

2 MGPOM Code

The coastal ocean is a region receiving a great deal of attention owing to an increased utilization for human habitation, aquatic development, and military operations. These activities require a knowledge of dynamic and thermodynamic structures of the coastal regions such as water circulation, ocean wave dynamics, storm surges, and evolution of seawater temperature and salinity. The POM is a three-dimensional, primitive equation, time-dependent, σ coordinate, free surface coastal ocean circulation model. The model primitive equations used describe the velocity, surface elevation, salinity, and temperature fields in the ocean. The ocean is assumed to be hydrostatic and incompressible (Boussinesq approximation).

Over the years, the traditional one-block rectangular grid has been used for ocean circulation modeling. This technology encounters difficulty on computational grids with high resolution owing to the large memory and processing requirements. For a large body of water with complicated coastlines, the number of grid points used in the calculation (water points) is often the same or even smaller than the number of unused grid points (land points). It is known that domain decomposition can be used to partition the traditional one-block grid into sub-domains that reduce the unused grid points and improve performance of the ocean model [9]. MPI [10] can be used to parallelize this type of computation by assigning each sub-block to a different MPI process. Communication is then used to synchronize data in the overlap regions of each block at appropriate places within the code.

A multiblock grid generation technique and parallel implementation of the MGPOM ocean circulation code was proposed and used effectively in [7]. The multiblock grid gen-

eration technique allows for the elimination of blocks composed mainly of land grid points. However, focusing on the elimination of land points from the new data set can result in blocks of widely varying size and the potential for severe load imbalance. For this study, we shall use threads to assist in restoring a better load balance between MPI processes. That is, rather than assign blocks to threads, threads shall be used in order to speed up execution within a block. Many of the same problems that would result from threading at the block level will be encountered by threading computations within blocks.

3 Pthreads and the Fortran API

Pthreads is the library of POSIX standard functions for concurrent, multithreaded programming. The POSIX standard only defines an application programming interface (API) to the C programming language, not to Fortran. Many scientific and engineering applications are written in Fortran. They would benefit from multithreading, especially on symmetric multiprocessors (SMP). In this section we present a brief background on multithreaded programming and the use of Pthreads. More complete descriptions of the POSIX thread library can be found in the books by Butenhof [2], Lewis and Berg [5], and Nichols et al [8]. We also present here some of the relevant details of the interface to that part of the Pthreads library that is compatible with standard Fortran.

3.1 Pthreads Details

Multithreading is a concurrent programming model. Multiple threads may execute concurrently on a uniprocessor system. Parallel execution, however, requires multiple processors sharing the same memory; i.e., SMP platforms. Threads perform concurrent execution at

the task or function level. A single process composed of independent tasks may break up these computations into a set of concurrently executing threads. Threads are created to execute an assigned subroutine within the program. Since all POSIX threads executing within a process are peers, there is no explicit parent-child relationship unless the programmer specifically implements such an association.

With separate threads executing within the same memory address space, there is the potential for memory access conflicts; i.e., multiple threads attempt to concurrently write to the same memory location (write/write conflict) or one thread is reading a memory location while another thread is concurrently writing to that same memory location (read/write conflict). Since scheduling of threads is largely non-deterministic, the order of thread operations may differ from one execution to the next. It is the responsibility of the programmer to recognize these potential conflicts and control them.

Pthreads provides a mechanism to control access to shared, modifiable data. Those portions of the code which allow access to shared data are known as *critical regions*. Locks, in the form of mutual exclusion (mutex) variables, prevent threads from entering critical regions of the program while the lock is held by another thread. Threads attempting to acquire a lock (i.e., enter a protected code region) will wait if another thread is already in the protected region.

Pthreads provides an additional form of synchronization through condition variables. Threads may pause execution until a particular condition has been met. (Where threads wait for the specific condition of a mutex to be released by another thread, condition variables allow a thread to wait on any feasible conditional expression the programmer requires.) The update of the shared variables within a conditional expression is protected

by a mutex associated with the condition variable. When a thread waits for the signal on the condition variable, the mutex is relinquished to allow other threads to affect the conditional expression being waited on. When a thread modifies variables of the conditional expression, a signal to wakeup may be sent to a thread waiting on the associated condition variable. The mutex is re-acquired after this wakeup signal is received. Because spurious wakeup signals are not disallowed by the POSIX standard and may be inadvertently received, good programming practice dictates the conditional expression be tested within a `WHILE` loop construct whose body contains a call to the condition variable wait routine. The conditional expression of this loop should evaluate to FALSE only when the condition being waited upon has been met, TRUE otherwise. Thus, if the wakeup condition has not been met, threads that receive incorrect signals will return to waiting since the while loop test will evaluate to TRUE.

3.2 FPTHRD Details

The FPTHRD package consists of a Fortran module and file of C routines. The module defines Fortran derived types, parameters, interfaces, and routines to allow Fortran programmers to use Pthread routines. The C functions provide the interface from Fortran subroutine calls and map parameters into the corresponding POSIX routines and function arguments.

The names of the FPTHRD routines are derived from the Pthreads root names; i.e., the string following the prefix **pthread_**. The string **fpthrd_** replaces this prefix. In this way, a call to the Pthreads function **pthread_create()** translates to a call to the Fortran subroutine **fpthrd_create()**. For consistency, all POSIX data types and defined constants

prefixed with **pthread_** (**PTHREAD_**) are defined with the prefix **fpthrd_** (**FPTHRD_**) within the Fortran module.

The Fortran API preserves the order of the arguments of the C functions and provides the C function value as the final argument. This trailing integer argument is most often used to return an indicator of the termination status of the routine. Fortran interface blocks also make it possible for the status parameter to be optional in all but one Fortran routine call.

An initial data exchange is required as a first program step before using other routines in FPTHRD. Initialization is performed with a call to the routine **fpthrd_data_exchange()**. This routine is similar in functionality to the **MPI_INIT()** routine from MPI. The data exchange was found to be necessary because the parameters defined in Fortran or constants defined in C are not directly accessible in the alternate language. One such value of note is the parameter **NULL** passed from Fortran to C routines. This integer is used as a signal within the C wrapper code to substitute a `NULL` pointer for the corresponding function argument.

4 How to Thread Existing Codes

Unlike OpenMP loop-level directives, Pthreads supports concurrency at the task or functional level. Thus, if entire subroutines from the code could be used for targets of thread creation, only minor modifications of the code would be necessary. Otherwise, the programmer's efforts to thread an existing code would be centered on locating those parts of the code which could be executed concurrently and extracting the lines from the program that

implement these portions into utility subroutines that are then used for thread creation and execution. It should be obvious that the former situation is more desirable.

With the goal of using the existing subroutine structure of the code as much as possible, we have found that threading existing codes involves three related phases:

1. Identify subroutines eligible for concurrent execution,
2. Locate concurrent execution within each subroutine, and
3. Insert calls to Pthreads routines to create and manage threads as well as enforce mutual exclusion required by data decomposition.

In the following sections, we shall examine each of these actions. Our experiences with threading the MGPOM code will be used for examples of each phase.

4.1 Identifying Subroutines

As stated previously, one of the goals of our threading efforts was to better load balance the execution time of those blocks that have more data than others. Previous studies [6] have used OpenMP for accomplishing this. Profiling the MGPOM code revealed several routines that accounted for more than half of the total execution time. The focus of our efforts was concentrated on these routines.

Once candidate subroutines are identified, each must be examined in more detail to ascertain whether or not the subroutine could be run concurrently within each MPI process. Methods for this determination are covered in more detail within the next section. The structure of the subroutine and use of data is directly related to how much labor will be involved in threading a chosen subroutine. In all cases, it is common sense that the amount

of effort required to thread a subroutine be compared to the potential amount of speedup that is expected to result. It makes little sense to put in many hours of work on a complex concurrency scheme within a subroutine that does not significantly affect the total amount of execution time.

4.2 Finding and Expressing Concurrency

After choosing subroutines for potential concurrent execution, each must be examined to determine if concurrent execution is feasible. For the MGPOM code, each subroutine is filled with many different loops operating on arrays from `COMMON` blocks and arrays sent as parameters. Thus, the potential for concurrency is dependent on how those arrays are partitioned and assigned to threads. Code written without loops may also provide concurrent execution potentials. Loops within a subroutine provide the easiest construct for locating potential concurrency within a subroutine. However, taking advantage of the identified concurrency may require drastic restructuring of the code such as encapsulation of concurrent functionality within new subroutines.

The most apparent case for finding concurrency is within loops where individual iterations may be executed independently. The loop iterations are simply divided among the created threads. Which iterations are assigned to which threads may be arbitrary or may be guided by some more orderly scheme based on the data. A static assignment of iterations is easy to implement with Pthreads since a schedule of work may be formulated onto a predetermined number of threads.

In order to implement a dynamic scheduling of iterations onto threads would involve keeping track of which threads have been assigned which iterations along with a mechanism

to assign new iterations to threads that have completed a previous iteration. This dynamic scheduling is not overly difficult to implement and would be useful in load balancing loop iterations that take different amounts of computation to complete.

In a more general case, subroutines may contain sequential code between loops. Once a method for breaking up loop iterations among threads is devised, it must be decided how to correctly execute the non-looped code. Because we are assuming that all threads execute the entire subroutine, this code may be executed concurrently with multiple threads or may be designated for execution by a single thread. The former case is most desirable since it would nominally involve innocuous duplication of local variables and a duplication of effort as all threads computed the same results. The latter case would require some mechanisms be coded in order to prohibit threads from executing code sequences that would be assigned to a single thread; the complexity of such mechanisms would depend upon the data decomposition and access restrictions to any global data structures used within the affected portions of code.

For the MGPOM code, we chose the static allocation model since all arrays used within all loops of the chosen subroutines had the same dimensionality (at least within the first two defined indices). A subroutine was written to determine the number of threads to be used within the MPI process based on the size of the data block assigned to the process and a fixed parameter value denoting the minimum threshold of grid points that would require a thread be created. A two-dimensional decomposition along the first two indices of the block (and consequently all other pertinent arrays) into sub-blocks is then computed: one sub-block per thread to be created. The indices within the block assigned to the process for each of the blocks is saved into a global array. These index values are used by each thread

created as loop iteration bounds within each threaded subroutine.

4.2.1 Finding Data Dependencies

While each loop was able to execute all iterations independently within the subroutines examined from MGPOM, this may not be the case with other codes. If there are data dependencies that restrict the order in which loop iterations must be executed, there may still be possibilities for concurrency within that loop. Such a situation would require a more careful synchronization of thread execution in order to enforce the correct iteration ordering.

Since there were no data dependencies within loops of the chosen MGPOM subroutines, we next checked for data dependencies between loops that resulted from the data decomposition. In this case, we were looking for potential read/write conflicts. That is, threads that access some array element that is within the assigned sub-block of another thread. For such a conflict the order of execution between the reading of an array value and the update of that array value must be done in the correct order. (It was determined that no write/write conflicts were possible in the threaded subroutine loops since all threads modified only those array elements assigned to them.)

In order to find any inter-loop dependencies between threads, a listing of the *read set* and *write set* of each loop was compiled and then compared. For our purposes with MGPOM, the read set of a loop is the set of all array elements that are used on the right hand side of an assignment statement (the value is "read" from memory) while the write set is the set of all array elements that are used on the left hand side of an assignment statement (the value is "written" into memory).

To identify quickly and completely all of the read and write sets internal to each loop contained within a subroutine a form was created. The form contained columns for loop numbers and the write set and read set for each loop. It is not enough to know just the names of the arrays within each set; the variables used to index these arrays within the loop are also needed. After all the data has been entered for each loop of a subroutine, the write set of each loop is compared to the read sets of all loops for any overlap.

The decomposition of data determines where overlap can occur. In the case of MGPOM, the data block assigned to each process that would create threads was divided along the first and second dimensions; i.e., the I and J axes of the array. Thus, overlap between threads under this static decomposition is possible when one thread accesses an array element outside the assigned sub-block. That is, an array reference within a loop contaning an index of `I+1`, `J+1`, `I-1`, or `J-1` has the potential of using a value "on the other side of the fence." Should a read (write) set contain a potential overlap index of an array contained within a write (read) set, there exists the potential for a read/write conflict and the order of execution between these loops must be preserved for correct execution. (It is possible to have read/write conflicts within the same loop. A more complex solution is required to handle these cases than is described below.)

As a concrete example of this process, consider the code extract of two loops from a subroutine of the MGPOM program shown in Figure 1. Figure 2 displays the form entries that detail the write and read sets for these loops. Since the data decomposition for the threaded version of MGPOM deals only with the first and second indices, we need only identify any overlap of array references containing index values of `I`$\pm$`1` and `J`$\pm$`1`. Examination of the data in Figure 2 shows that such an overlap exists, specifically, `A(I+1,J,K)` or

Figure 1 MGPOM Code Example for Read/Write Conflicts

```
      DO 315 K=2,KBM1
      DO 315 J=1,JM
      DO 315 I=1,IM
        A(I,J,K)=A(I,J,K)
     &          -.50*(AAM(I,J,K)+AAM(I-1,J,K))*(H(I,J)+H(I-1,J))
     &          *(QB(I,J,K)-QB(I-1,J,K))*DUM(I,J)/(DX(I,J)+DX(I-1,J))
        C(I,J,K)=C(I,J,K)
     &          -.50*(AAM(I,J,K)+AAM(I,J-1,K))*(H(I,J)+H(I,J-1))
     &          *(QB(I,J,K)-QB(I,J-1,K))*DVM(I,J)/(DY(I,J)+DY(I,J-1))
        A(I,J,K)=.50*(DY(I,J)+DY(I-1,J))*A(I,J,K)
        C(I,J,K)=.50*(DX(I,J)+DX(I,J-1))*C(I,J,K)
  315 CONTINUE

      DO 230 K=2,KBM1
      DO 230 J=1,JM
      DO 230 I=1,IM
         QF(I,J,K)=(W(I,J,K-1)*Q(I,J,K-1)
     &             -W(I,J,K+1)*Q(I,J,K+1))/(DZ(K)+DZ(K-1))*ART(I,J)
     &             +A(I+1,J,K)-A(I,J,K)+C(I,J+1,K)-C(I,J,K)
         QF(I,J,K)=((H(I,J)+ETB(I,J))*ART(I,J)*
     &              QB(I,J,K)-DT2*QF(I,J,K))/((H(I,J)+ETF(I,J))*ART(I,J))
  230 CONTINUE
```

`C(I,J+1,K)` of the read set for loop 230 overlap with `A(I,J,K)` and `C(I,J,K)` of the write set of loop 315.

If loops are separated from one another by several other loops or intervening lines of code, it might be assumed that the correct execution order will naturally occur. This is not necessarily the case. As stated previously, the order of execution for concurrent threads is non-deterministic and the actual execution order between threads cannot be predicted. Good programming practice requires that even when the slightest potential for some conflict to occur is present, steps must be taken to specifically ensure a correct execution ordering.

There are several methods available within the functionality of Pthreads or that can be constructed with Pthreads routines to coordinate execution between threads. In order to

Figure 2 Write Set and Read Set Form for Loops 315 and 230

Loop	Write Set	Read Set
315	A(I,J,K) C(I,J,K)	A(I,J,K) AAM(I,J,K) AAM(I-1,J,K) H(I,J) H(I-1,J) QB(I,J,K) QB(I-1,J,K) DUM(I,J) DX(I,J) DX(I-1,J) C(I,J,K) AAM(I,J-1,K) H(I,J-1) DVM(I,J) DY(I,J) DY(I,J-1) DY(I-1,J) DX(I,J-1)
230	QF(I,J,K)	W(I,J,K-1) Q(I,J,K-1) W(I,J,K+1) Q(I,J,K+1) DZ(K) DZ(K-1) ART(I,J) A(I+1,J,K) A(I,J,K) C(I,J+1,K) C(I,J,K) H(I,J) ETB(I,J) QB(I,J,K) DT2 QF(I,J,K) ETF(I,J)

preserve simplicity within the threaded MGPOM code, we chose to use a *barrier* placed between loops that had potential for read/write conflicts. A description of the barrier implementation used is given in the next section.

4.2.2 Barriers

Barriers are constructs that halt execution of threads until all threads have reached the barrier. Once all threads have reached the barrier position within the code, they are released to continue execution. Barriers are, thus, a synchronization point for all threads. The module code for the barrier implementation used within MGPOM is given in Appendix A. This code is a Fortran version of the barrier code written in C found in [2].

The barrier derived type (**BARRIER_T**) contains a mutex, a condition variable, three integers and a `LOGICAL` toggle. The integers are used to denote whether a barrier instance has been properly initialized, to keep track of the number of threads that must reach the barrier before the threads are released, and to hold a count of the number of threads that are currently waiting at the barrier. The mutex controls access to the integer counts and the toggle as well as protect the condition variable. The condition variable is used to put threads

to sleep that have reached the barrier and also as a mechanism to awaken those threads for continuation of execution when the last thread arrives at the barrier. The toggle is used in the conditional expression that ultimately allows threads to proceed from the barrier.

There are three routines within the barrier module. **BARRIER_INIT()** initializes the barrier and validates it. This routine includes the default initialization of the mutex and condition variable as well as setting the number of threads that must reach the barrier in order to trigger the release of all threads held. The **BARRIER_DESTROY()** routine uses Pthread functions to destroy the mutex and condition variable and invalidate the barrier for future use.

When each thread arrives at the barrier call, **BARRIER_WAIT()**, the mutex is acquired and the count is decremented. If the count is not zero, the current value of the toggle is copied into a local variable and the thread is put to sleep by calling **fpthrd_cond_wait()** (which also releases the mutex). By comparing the local copy of the toggle value with the global barrier toggle value, threads which might be inadvertently woken up before the last thread has arrived will be put back to sleep. When the final thread needed to reach the barrier arrives, the count is decremented to zero. This final thread switches the value of the toggle, resets the count for the next use of the barrier, and broadcasts a wakeup signal to all other held threads. Upon wakeup from the broadcast signal, each thread will check the value of the global toggle to their local copy, determine that the two values are different, and proceed to the code following the barrier call.

4.3 Pthread Calls

The final phase for the threading of existing codes is to insert calls to the FPTHRD subroutines. Above we have described how some calls were encapsulated within the barrier implementation. Also, it is hoped that any other calls to synchronization routines needed to ensure correct execution would be placed as needed. The other major chore that needs to be completed is the insertion of code to create threads that will execute the threaded subroutines.

The thread creation routine allows a single argument to be sent to the subroutine. If the original subroutine that is to be threaded uses more than a single parameter, some adjustments need to be made. It is recommended that all parameters to the target subroutine be placed within a global module that can be USE-associated within the subroutine and the calling routine. This would allow the single parameter to be used to send an integer to the subroutine that would contain a unique thread number. Within the MGPOM code, this unique thread number is used to index the global index array for the loop bounds computed via the data decomposition subroutine. One other possibility would be to create a derived type that holds all the different parameter values (as well as the thread number, if needed) required by the threaded subroutine. In any event, some modification of the subroutine header and handling of parameters will be necessary.

The above is easily applied to subroutines that are called at a single point within the overall code. However, it is common practice to employ a subroutine several times within a code for performing the same computations on different parameter sets. In order to thread such a subroutine, a more involved code transformation is needed. In this instance, as

before, all parameters should be encapsulated within a module for thread access. Where the subroutine header was modified above, a number of dummy subroutines are written which accept a single parameter. It is these dummy subroutines that are used in thread creation and their only function is to call the target subroutine with the appropriate set of parameters.

For example, assume subroutine A is to be executed concurrently and is called from three different points within a program with three different sets of parameters. All three sets of parameters are defined within a module (or three separate modules dependent upon code requirements) and three dummy subroutines, say A1, A2, and A3, are created. Each of the different dummy routines simply contains a call to subroutine A with one of the original parameter sets. When creating threads for each individual call to subroutine A, the threads are created using the appropriate dummy subroutine.

If the code contains consecutive calls to the same routine with different parameter sets, a single dummy subroutine can be constructed that calls the subroutine with each different data set. Thus, the overhead of creating threads for multiple subroutines is reduced to a single instance. Agglomerating any number of consecutive threaded subroutines can be done within a single dummy subroutine. There is no need for the subroutines to be the same. However, because some threads may complete execution of one call to a subroutine before others, the programmer must ensure that there are no data dependencies between different calls to the routines called within the dummy subroutine. A barrier call between subroutine calls would delay execution of a subsequent routine until all threads had completed execution of the prior subroutine.

The code to create threads at the calling point of the threaded subroutine may simply

be a loop over the number of threads to be created that calls the **fpthrd_create** routine. In the most simple case, following this loop would be another loop to join all the created threads. This second loop would pause the creating thread until all created threads had finished. Other activities can be pursued by the creating thread, including taking a share of the work to be done. While this does reduce the amount of thread resources that would be used, programming for any other activity of the creating thread will require more complex coding.

5 Performance of Threaded POM

The physical geographic area we chose to run for this study is the Persian Gulf. This area extends from 48 East to 58 East in longitude and from 23.5 North to 30.5 North in latitude. Part of the Gulf of Oman is also included in this physical domain. The twenty-block grid contains a total of 32,031 grid points with only 9,722 of those as unused land points. The twenty-block grid was generated from a one-block grid by a simple algebraic scheme using the EAGLEView software package [11]. Details of the grid generation techniques used to create this multiblock grid data set can be found in [7].

All runs reported in this section were performed on an SGI Origin 2000. These runs computed a 10-day simulation of the Persian Gulf model. The MPI-only version of MGPOM took 4325 seconds ($\sim$1.2 hours) using 20 processors.

The first subroutine chosen for threading was PROFQ which was found to be the dominant subroutine with regard to execution time during the profiling of the MGPOM code. The PROFQ-threaded version of MGPOM took 3354 seconds (55.9 minutes) to run the

10-day Persian Gulf simulation. This represents a 22.5% reduction in execution time for changes that took less than one half hour to make the code modifications.

To date, four of the longest executing subroutines from the MGPOM code have been threaded. This version of the code runs the 10-day simulation in 2835 seconds (47.25 minutes) or a 34.5% reduction of wallclock execution time over the MPI-only MGPOM code.

For all threaded code runs, a total of 44 processors were requested from which 20 were used to run the MPI processes. It is assumed that the threads created during the threaded MGPOM runs were migrated to the extra processors allocated to the run.

6 Conclusion

We have presented the methods used to convert an ocean circulation model code for multithreaded execution using the Fortran 90 API to Pthreads developed at the U.S. Army Engineer Research and Development Center Major Shared Resource Center. We have also demonstrated that this threaded code runs faster than the original version.

The techniques described herein should be applicable to a large number of other scientific codes. With some threads programming experience, it is felt that programmers would be able to develop more complex threaded codes from current Fortran codes.

References

[1] A. F. Blumberg and G. L. Mellor. A Description of a Three-Dimensional Coastal Ocean Circulation Model. *Three-Dimensional Coastal Models*, 1, 1987.

[2] David R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, Reading, MA, 1997.

[3] Richard J. Hanson, Clay P. Breshears, and Henry A. Gabb. A Fortran Interface to POSIX Threads. Technical Report 00-18, ERDC MSRC/PET, 2000.

[4] *9945-1:1996 (ISO/IEC) [IEEE/ANSI Std 1003.1 1996 Edition] Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language] (ANSI)*, IEEE Standards Press, 1996.

[5] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads.* Sun Microsystems Press, Mountain View, CA, 1998.

[6] P. V. Luong, C. P. Breshears, and H. A. Gabb. Execution and Load-Balance Improvements in the CH3D Hydrodynamic Simulation Code. Technical Report 00-07, ERDC MSRC/PET, February 2000.

[7] Phu Luong, Clay P. Breshears, and Le N. Ly. Dual-Level Parallelism and Multiblock Grids in Coastal Ocean Circulation Modeling. Technical Report 00-08, ERDC MSRC/PET, 2000.

[8] Bradford Nichols, Dick Buttlar, and Jacqueline Prolux Farrell. *Pthreads Programming.* O'Reilly and Associates, Sebastopol, CA, 1996.

[9] W. D. Oberpriller, A. C. Sawdey, M. T. O'Keefe, and S. Gao. Parallelizing the Princeton Ocean Model Using TOPAZ. `http://topaz.lcse.umn.edu`.

[10] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI—The Complete Reference: Volume 1, the MPI Core.* The MIT Press, Cambridge, MA, 1998.

[11] M. Stokes, M. Jiang, and M. Remotique. *EAGLEview Grid Generation Package, EAGLEView Version 2.4 Manual.* Missisippi State University/National Science Foundation Engineering Research Center for Computational Field Simulation, December 1992.

APPENDIX A—FPTHRD Barrier Code

```
MODULE BARRIER
!=====================================================================!
!  Module for F90 Pthreads API to define a barrier for threads
!
!  Written by Clay Breshears, 7 FEB 2000
!
!  Based on Butenhof "Programming with POSIX Threads," Section 7.1.1
!=====================================================================!

   USE FPTHRD

   TYPE BARRIER_T
      TYPE(FPTHRD_MUTEX_T) :: mutex      ! control access to barrier
      TYPE(FPTHRD_COND_T) ::  cv         ! wait for barrier
      INTEGER ::              valid      ! set when valid
      INTEGER ::              threshold  ! number of threads required
      INTEGER ::              counter    ! current number of threads
      LOGICAL ::              cycle      ! alternate cycles (T or F)
   END TYPE BARRIER_T

   INTEGER, PARAMETER, PRIVATE:: BARRIER_VALID = 14404350  ! 0xdbcafe

CONTAINS

   SUBROUTINE BARRIER_INIT(B, C, STATUS)
   !
   !  Initialize a barrier for use
   !
      TYPE(BARRIER_T), INTENT(OUT):: B
      INTEGER, INTENT(IN):: C
      INTEGER, INTENT(OUT):: STATUS

      INTEGER:: ierr
      TYPE(C_PTR) NULL

      NULL=C_NULL

      B%threshold = C
      B%counter = C
      B%cycle = .FALSE.
      CALL FPTHRD_mutex_init(B%mutex, NULL, STATUS)
      IF (STATUS .NE. 0) RETURN
```

```
      CALL FPTHRD_cond_init(B%cv, NULL, STATUS)
      IF (STATUS .NE. 0) THEN
         CALL FPTHRD_mutex_destroy(B%mutex, ierr)
         RETURN
      ENDIF
      B%valid = BARRIER_VALID
      RETURN
   END SUBROUTINE BARRIER_INIT

   SUBROUTINE BARRIER_DESTROY(B, STATUS)
   !
   !  Destroy a barrier when done using it
   !
      TYPE(BARRIER_T), INTENT(INOUT):: B
      INTEGER, INTENT(OUT):: STATUS

      INTEGER:: ierr

      IF (B%valid .NE. BARRIER_VALID) THEN
         STATUS = EINVAL
         RETURN
      ENDIF

      CALL FPTHRD_mutex_lock(B%mutex, STATUS)
      IF (STATUS .NE. 0) RETURN
   !
   !  Check whether any threads are known to be waiting;  report
   !    "BUSY" if so
   !
      IF (B%counter .NE. B%threshold) THEN
         CALL FPTHRD_mutex_unlock(B%mutex, STATUS)
         STATUS = EBUSY
         RETURN
      ENDIF

      B%valid = 0
      CALL FPTHRD_mutex_unlock(B%mutex, STATUS)
      IF (STATUS .NE. 0) RETURN
   !
   !  If unable to destroy either mutex or cond_var object,
   !    return the error status
   !
      CALL FPTHRD_mutex_destroy(B%mutex, STATUS)
      CALL FPTHRD_cond_destroy(B%cv, ierr)
      IF (STATUS .EQ. 0) STATUS = ierr
```

```
   RETURN
END SUBROUTINE BARRIER_DESTROY

SUBROUTINE BARRIER_WAIT(B, STATUS)
!
!  Wait for all members of a barrier to reach the barrier.  When
!    the count (of remaining members) reaches 0, broadcast to wake
!    all threads waiting.
!
   TYPE(BARRIER_T), INTENT(INOUT):: B
   INTEGER, INTENT(OUT):: STATUS

   INTEGER:: CANCEL, TMP, ierr
   LOGICAL:: CYCLE

   IF (B%valid .NE. BARRIER_VALID) THEN
      STATUS = EINVAL
      RETURN
   ENDIF

   CALL FPTHRD_mutex_lock(B%mutex, STATUS)
   IF (STATUS .NE. 0) RETURN

   CYCLE = B%cycle                  ! Remember which cycle we're on

   B%counter = B%counter - 1
   IF (B%counter .EQ. 0) THEN
      B%cycle = .NOT. B%cycle
      B%counter = B%threshold
      CALL FPTHRD_cond_broadcast(B%cv, STATUS)
      !
      !    The last thread into the barrier will return status
      !      -1 rather than 0, so that it can be used to perform
      !      some special serial code following the barrier
      !
      if (STATUS .EQ. 0) STATUS = -1

   ELSE
      !    Wait with cancellation disabled, because BARRIER_WAIT
      !      should not be a cancellation point.
      !
      CALL FPTHRD_setcancelstate(PTHREAD_CANCEL_DISABLE, CANCEL, ierr)

      !    Wait until the barrier's cycle changes, which means
      !      that it has been broadcast, and we don't want to wait
```

```
      !      anymore.
      !
      DO WHILE (CYCLE .EQV. B%cycle)
         CALL FPTHRD_cond_wait(B%cv, B%mutex, STATUS)
         IF (STATUS .NE. 0) EXIT
      END DO

      CALL FPTHRD_setcancelstate(CANCEL, TMP, ierr)
   ENDIF

   !    Ignore an error in unlocking.  It shouldn't happen, and
   !      reporting it here would be misleading -- the barrier wait
   !      completed, after all, whereas returning, for example,
   !      EINVAL would imply the wait had failed.  The next attempt
   !      to use the barrier *will* return an error, or hang, due
   !      to whatever happened to the mutex.
   !
   CALL FPTHRD_mutex_unlock(B%mutex, ierr)
   RETURN
END SUBROUTINE BARRIER_WAIT
```